Lean

PENG

Created: 20260910, Version: 20260910

Contents

1 Introduction

`https://lean-lang.org/functional_programming_in_lean/`

Lean is an **expression-oriented** functional language with dependent types.

1.1 Ealuating Expressions

- every expression must have a type before it can be evaluated
- once given a value, variables cannot be reassigned
- parentheses are necessary when a function's argument is to be computed via another function call, i.e. $5 * (1 + 2)$

1.2 Types

Definition 1.1 (type)**.**

- `Nat`*: default type for non-negative integer literals*
- `Int`
- `Float`
- `String`
- `Bool`
- `List`*:* `List type`
- `Type`*: describes other types, so* `Nat`*,* `String`*,* `List`*, etc. all have type* `Type`

Example 1.2.

```
#eval (1 - 2 : Nat) -- 0
#check (1 - 2 : Int) -- check the type of an expression
```

1.3 Functions and Definitions

Definition 1.3 (definition)**.**

```
def var : type := val
```

Example 1.4.

```
def lean : String := "Lean"
#eval String.append "hello " lean
```

1.3.1 Defining Functions

Definition 1.5 (function)**.**

```
def func (arg1 : type) ... (argn : type) : type := body
```

Example 1.6.

```
def maximum (n : Nat) (m : Nat) : Nat := if n < m then m else n
#eval maximum 1 5
```

Definition 1.7 (currying)**.** `Nat -> (Nat -> Nat)`

1.3.2 Defining Types

Definition 1.8 (Type)**.**

```
def Str : Type := String -- overloaded
abbrev N : Type := Nat -- unfolded
```

Definition 1.9 (overloaded)**.**

Definition 1.10 (reducible)**.** *Definitions that are to be unfolded are called reducible.*

1.4 Structures

Definition 1.11 (structure)**.**

```
structure Point where
x : Float
y : Float
def origin : Point := {x:=0.0,y:=0.0}
#eval origin.x
#check {x:=0.0,y:=0.0:Point}
```

Definition 1.12 (updating structures)**.** `def zeroX (p : Point) := {p with x:=0}`

Definition 1.13 (constructor)**.**

```
structure Point where
point::
x : Float
y : Float
#check Point.point 1.0 2.0
```

Definition 1.14 (accessor)**.**

```
#eval Point.x {x:=1.0,y:=2.0}
#check {x:=1.0,y:=2.0 : Point}.x
-- TARGET.func ARG1 ... ARGn
#eval "hello".append " Lean"
def Point.modifyBoth (f : Float -> Float) (p : Point) : Point := {x:=f p.x,y:=f p.y}
#eval {x:=3,y:=4 : Point}.modifyBoth Float.floor
```

1.5 Datatypes and Patterns

Definition 1.15 (datatype)**.**

- `sum`
- `recursive`
- `inductive` *: recursive sum type*

Example 1.16 (inductive)**.**

```
inductive Nat where
| zero : Nat
| succ (n : Nat) : Nat
#eval Nat.succ (Nat.succ Nat.zero)
```

1.5.1 Pattern Matching

Definition 1.17 (pattern matching)**.**

```
def isZero (n:Nat) : Bool :=
match n with
| Nat.zero => true
| Nat.succ k => false
#eval isZero Nat.zero

def pred (n : Nat) : Nat :=
match n with
| Nat.zero => Nat.zero
| Nat.succ k => k
#eval pred 5

def depth (p:Point3D) :Float :=
match p with
| {x:=h,y:=w,z:=d} => d
#eval depth {x:=1,y:=2,z:=3}
```

1.5.2 Recursive Functions

Definition 1.18 (recursive definition)**.** *Definitions that refer to the name being defined are called recursive definition.*

Definition 1.19 (structural recursion)**.**

```
def even (n:Nat) : Bool :=
match n with
| Nat.zero => true
| Nat.succ k => not (even k)
#eval even 2

-- addition = interated Nat.succ
def plus (n:Nat) (k:Nat) : Nat :=
match k with
| Nat.zero => n
| Nat.succ k' => Nat.succ (plus n k')
#eval plus 1 2

-- multiplication = interated addition
def times (n:Nat) (k:Nat) : Nat :=
match k with
| Nat.zero => 0
```

```
| Nat.succ k' => plus n (times n k')
#eval times 1 2

-- subtraction = interated predecessor
def minus (n:Nat) (k:Nat) : Nat :=
match k with
| Nat.zero => n
| Nat.succ k' => Nat.pred (minus n k')
#eval minus 2 1
```

Remark 1.20. *Lean ensures by default that every recursive function will eventually reach a base case to avoid accidental infinite loops*

Example 1.21.

```
def div (n:Nat) (k:Nat) : Nat :=
if n < k then
0
else Nat.succ (div (n-k) k)
```

1.6 Polymorphism

Definition 1.22 (polymorphism)**.** *polymorphism refers to datatypes and definitions that take types as arguments, where* ***types*** *are ordinary* ***expressions*** *in Lean.*

Example 1.23.

```
-- structure take types as arguments
-- a for \alpha
structure PPoint (a : Type) where
x : a
y : a
def natOrigin : PPoint Nat := {x:=Nat.zero,y:=Nat.zero}
#check natOrigin

-- definition take types as arguments
def replaceX (a : Type) (point : PPoint a) (newX : a) : PPoint a :=
{point with x:=newX}
#check replaceX
#eval replaceX Nat natOrigin 2

-- argument is a sign
inductive Sign where
| pos
| neg
def posOrNegThree (s : Sign) :
match s with | Sign.pos => Nat | Sign.neg => Int :=
match s with
| Sign.pos => (3:Nat)
| Sign.neg => (-3:Int)
#check posOrNegThree Sign.pos
```

Remark 1.24. *It is customary to use Greek letters to name type arguments.*

1.6.1 Linked List Datatype

Definition 1.25 (List)**.** *A* `List a` *can be built with either* `nil` *or* `cons`*. A list that contains n entries contains n* `cons` *constructors, the last of which has* `nil` *as its tail.*

```
inductive List (a : Type) where
| nil : List a
```

```
| cons : a -> List a -> List a
def primeUnder10 : List Nat := [2,3,5,7]
#eval List.cons 2 (List.cons 3 (List.cons 5 (List.cons 7 List.nil)))

-- functions that consume Lists
def length (a : Type) (xs : List a) : Nat :=
match xs with
| List.nil => Nat.zero -- [] for List.nil
| List.cons y ys => Nat.succ (length a ys) -- y :: ys for List.cons
#eval length String ["hello", "lean"]
```

1.6.2 Implicit Arguments

Definition 1.26 (implicit type)**.**

```
def length {a : Type} (xs : List a) : Nat :=
match xs with
| [] => Nat.zero
| y :: ys => Nat.succ (length ys)
#eval length ["hello", "lean"]
-- List.length
#eval ["hello", "lean"].length
```

1.6.3 More Built-In Structures and Inductive Datatypes